Available online at www.sciencedirect.com
ScienceDirect
Discrete Optimization 4 (2007) 40–62
DISCRETE OPTIMIZATION
www.elsevier.com/locate/disopt

Symmetric ILP: Coloring and small integers

François Margot [1]

Tepper School of Business, Carnegie Mellon University, Pittsburgh, PA 15213-3890, United States

Received 21 July 2005; received in revised form 14 September 2006; accepted 6 October 2006
Available online 22 December 2006

Abstract

This paper presents techniques for handling symmetries in integer linear programs where variables can take integer values, extending previous work dealing exclusively with binary variables. Orthogonal array construction and coloring problems are used as illustrations.

MSC: 90C10; 90C27; 90C57

Keywords: Branch-and-cut; Isomorphism pruning; Symmetry

1. Introduction

An integer linear program (ILP) is *symmetric* if its variables can be permuted without changing the structure of the problem. Symmetric ILP frequently appear when formulating classical problems in combinatorics or optimization and are difficult to solve with traditional branch-and-cut techniques. (We assume that the reader is familiar with these procedures, as excellent introductions can be found in [8,10,23,26].) For more background material related to these problems, the reader is referred to [18,19].

The present paper generalizes the approach of [19] to ILP having general integer variables. Reformulating such problems as binary problems is of course possible but solving such a reformulation might be very inefficient due to the increase in problem size: For example, an integer variable x_j with bounds $0 \leq x_j \leq p$ may be replaced by p binary variables. These p variables are equivalent, implying that the order of the symmetry group of the original problem is multiplied by $(p!)$. Doing so for 100 variables results in an ILP with $100p$ variables and a group whose order is multiplied by $(p!)^{100}$. Note that even when the ILP is not symmetric, it is known that replacing integer variables by collections of binary variables is inadvisable. For example, Owen and Mehrotra [22] show that these reformulations usually require a larger branch-and-bound tree than the original one. They also show that convexification procedures will be weaker when reformulations are used. Avoiding the multiplication of variables is thus attractive. The approach taken in this paper is to record the variables with positive value and their value separately. The group operations are then performed with the original group, with additional conditions attached to the recorded values.

E-mail address: fmargot@andrew.cmu.edu.
[1] Work supported by ONR grant N00014-03-1-0188.

1572-5286/$ - see front matter © 2006 Elsevier B.V. All rights reserved.
doi:10.1016/j.disopt.2006.10.008

We also consider problems that put "colors" on "objects". These problems are usually difficult to solve using ILP when, in addition to the symmetry between the colors, the "objects" have symmetries among themselves. Here also, we are interested in devising an algorithm that treats both types of symmetries together, but taking advantage of the simple symmetry between the colors in a more efficient way than encoding it in the symmetry group of the problem.

Results in this paper are a generalization and strengthening of the results from [18,19] where a branch-and-cut algorithm is devised for solving binary ILP with large symmetry groups. Notations and basic definitions are in Section 2. Section 3 is a straightforward extension of the *ranked branching rule* of [19] to the non binary case. The remainder of the paper is split into two, Section 4 dealing with the case of an ILP with general integer variables and Section 5 treating the case of coloring problems. The main result of this paper is Lemma 6 in Section 4. It forms the basis of a procedure to exclude values for some variables that is very similar to domain reduction techniques in constraint programming. Section 4 also presents a comparison between three different formulations for ILP with general integer variables. The example problem is the construction of orthogonal arrays, a classical problem in combinatorial design having a natural ILP formulation with variables that can take small integer values.

Section 5 presents a comparison between the branch-and-cut of [19], the branch-and-cut of Section 4 and the one of Section 5 for solving edge coloring problems on graphs having a non trivial automorphism group.

2. Preliminaries

Let Π^n be the set of all permutations of the ground set $I^n = \{1, \ldots, n\}$. Π^n is known as the symmetric group of I^n. A permutation in Π^n is represented by an n-vector π, with $\pi[i]$ being the image of i under π. If v is an n-vector and $\pi \in \Pi^n$, let $w = \pi(v)$ denote the vector w obtained by permuting the coordinates of v according to π, i.e.,

$$w[\pi[i]] = v[i] \quad \text{for all } i \in I^n.$$

To simplify the notation, we make no difference between a set $S \subseteq I^n$ and its characteristic vector. Hence $\pi(S)$ is the subset of I^n containing $\pi[i]$ for all $i \in S$.

Let $K = \{0, 1, \ldots, k\}$ for some positive integer value k. We consider an integer linear program (ILP) of the form

$$\begin{aligned} &\min \quad c^T x \\ &\text{s.t. } Ax \geq b, \\ &x \in K^n, \end{aligned} \tag{1}$$

where A is an $m \times n$ matrix. Let Q be the set of all feasible solutions of the ILP. The *symmetry group* G of this ILP is the set of all permutations π of the n variables mapping Q on itself and mapping each feasible solution on a feasible solution having the same value, i.e.

$$G = \{\pi \in \Pi^n \mid c^T \bar{x} = c^T \pi(\bar{x}) \text{ and } \pi(\bar{x}) \in Q \text{ for all } \bar{x} \in Q\}.$$

Note that in most situations G is not known, but a subgroup G' of G is. All the results in this paper hold if G is replaced by G', but it should be expected that the pruning obtained with G' will be weaker than the one obtained with G.

The *orbit* of $S \subseteq I^n$ under G is

$$\text{orb}(S, G) = \{S' \subseteq I^n \mid S' = g(S) \text{ for some } g \in G\}.$$

The *stabilizer* of S in G is the subgroup of G given by:

$$\text{stab}(S, G) = \{g \in G \mid g(S) = S\}.$$

We are, most of the time, dealing with a subset $S \subseteq I^n$, each variable with index in S having a specified value in K. If v is a vector such that $v[i]$ is the value associated with x_i for all $i \in S$ and $v[i] = -1$ for all $i \notin S$, then the pair (S, v) is called a *valset*. We extend the definitions of orbit and stabilizer to valsets as follows:

If (S, v) is a valset, its *orbit* under G is

$$\text{orb}(S, v, G) = \{(S', v') \mid S' = g(S) \text{ and } v' = g(v) \text{ for some } g \in G\}.$$

The *stabilizer* in G of a valset (S, v) is the subgroup of G given by:

$$\text{stab}(S, v, G) = \{g \in G \mid g(S) = S \text{ and } g(v) = v\}.$$

For any n-vector w and for $1 \leq a \leq b \leq n$, we write $w[a..b]$ for the entries $\{w[a], w[a+1], \ldots, w[b]\}$ of w as an unordered set.

If $g_1, \ldots, g_s$ are s permutations of I^n, the permutation $g = g_1 \cdot \cdots \cdot g_s$ is obtained by applying the permutations from right to left, i.e. $g(w) = g_1(g_2(\ldots(g_s(w))\ldots))$ for any n-vector w.

The proposed branch-and-cut algorithm will branch by fixing the value of one variable x_j to values in K. We make a difference between a variable *fixed* to value q and a variable *set* to q: A variable is fixed to q if this is the result of a branching operation; it is set to q if, for some reason other than a branching decision (e.g. reduced cost fixing, logical implications), the variable must take that particular value.

In many combinatorial problems expressed as an ILP, the number of variables having a positive value in an optimal solution is a small fraction of all the variables. Treating variables having value 0 a little bit differently than variables having a positive value might thus allow for more efficient algorithms. This is the justification for the following definitions. Let a be a node of the branch-and-cut enumeration tree. We denote by F_0^a (resp. F_p^a) the set of indices of variables fixed to 0 (resp. to a positive value) at a. We use S_0^a (resp. S_p^a) for the set of indices of variables set to 0 (resp. to a positive value) at a. We denote by F^a (resp. N^a) the set of indices of variables that are fixed (resp. not fixed) at a. Note that $S_0^a \cup S_p^a \subseteq N^a$. We also define the n-vectors f_p^a (resp. fs_p^a) such that (F_p^a, f_p^a) (resp. $(F_p^a \cup S_p^a, fs_p^a)$) is the valset corresponding to the fixing (resp. to the fixing and setting) to positive values at a.

3. Branching rule

Let a and b be two nodes of the enumeration tree of a branch-and-cut algorithm. The subproblems associated with nodes a and b of the branch-and-cut are isomorphic if there exists a permutation $g \in G$, such that $g(F^a) = F^b$ and for each $i \in F^a$, the value of x_i in a is the same as the value of $x_{g(i)}$ in b.

As explained in [18], this definition is difficult to use for an efficient pruning of the tree. Following principles of isomorphism pruning in combinatorial algorithms [5,14,20], a practical pruning scheme is devised based on the concept of a *representative* for subproblems in an isomorphism class. Provided that the branching rule satisfies some condition, all isomorphic subproblems that are not representative of their class can be pruned. This section is a straightforward extension to valsets of results in [19]. Most proofs are omitted since they essentially mirror the proofs in that paper.

During the branch-and-cut, we maintain a vector R of n integers, the *rank vector*, indicating the order in which the variables have been used as branching variables: At the beginning, $R[i] = n + 1$ for $i = 1, \ldots, n$ and $r = 0$. If variable x_h is chosen for branching and $R[h] = n + 1$, then $R[h]$ is set to $r + 1$ and r is increased by one. Note that both R and r are global variables (i.e., the same R and r are in use at each node of the enumeration tree) and that r is never decreased during the whole enumeration.

The rule to select the branching variable x_h at a, called the *ranked branching rule*, is then the following:

(i) If there exists $j \in N^a$ with $R[j] < n + 1$, then $h = \text{argmin}\{R[j] \mid j \in N^a\}$.
(ii) Otherwise, choose freely any index $h \in N^a$.

It follows that if each variable has been used at least once as a branching variable, the resulting rank vector R is a permutation of I^n. The ranked branching rule is a little more flexible than the *minimum index branching rule* of [18]. The latter amounts to always choosing the minimum index in N^a in step (ii) of the ranked branching rule, or, equivalently, assuming from the start that $R[j] = j$ for all $j \in N$.

Using the ranked branching rule makes some results a little bit more difficult to present than if the minimum index branching rule was used, but the annoyed reader can always assume that R is initialized such that $R[j] = j$ for all j if he wants to get the results for the latter.

Note that a variable x_j with $j \in S^a$ might be the chosen branching variable. Then the rank vector is updated, a unique son b is created (fixing x_j to the value it is currently set to), and variable x_j becomes one of the fixed variables.

In the remainder of the paper, we consider a branch-and-cut using a ranked branching rule. The rank vector R at the start of the processing of node a is denoted by R^a. R^a depends on the enumeration strategy, but the results given below are valid for any enumeration strategy.

Let (J, v) and (J', v') be two valsets. We say that (J, v) is *lexicographically smaller or equal to* (J', v') with respect to a rank vector R, written $(J, v) \preceq (J', v')$, if the following condition is satisfied:

Order the elements in J as $(j_1, \ldots, j_s)$ according to non-decreasing value of their rank, and then, for indices with identical rank, according to non-increasing value of $v[j_k]$. Order the elements in J' as $(j'_1, \ldots, j'_t)$ in a similar way. Then, there exists $u \in \{1, \ldots, s+1\}$ such that $R[j_i] = R[j'_i]$ and $v[j_i] = v'[j'_i]$ for all $i \in \{1, \ldots, u-1\}$ and one of the following holds:

(i) $u = s + 1$;
(ii) $R[j_u] < R[j'_u]$;
(iii) $R[j_u] = R[j'_u]$ and $v[j_u] > v'[j'_u]$.

We say that (J, v) is *lexicographically smaller than* (J', v') with respect to a rank vector R, written $(J, v) \prec (J', v')$, if (J, v) is lexicographically smaller or equal to (J', v') with respect to R and either condition (i) holds and $s < t$ or condition (ii) or (iii) holds.

For a given rank vector R, a valset (J, v) is a *representative* of the valsets in its orbit under G if (J, v) is lexicographically smaller or equal to any other valset in its orbit under G, i.e.

$$(J, v) \preceq (g(J), g(v)) \quad \forall g \in G.$$

Notice that, for any rank vector R, there is at least one representative in the orbit of J and, possibly, more than one.

Lemma 1. *Let R_1 and R_2 be two rank vectors obtained during a branch-and-cut using a ranked branching rule and assume that R_2 is obtained after R_1. Then*

(i) *if (J, v) is not a representative with respect to R_1 then (J, v) is not a representative with respect to R_2*
(ii) *if (J, v) is a representative with respect to R_1 and all the entries $R_1[j]$ for $j \in J$ are strictly smaller than $n + 1$ then (J, v) is the unique representative of its orbit with respect to R_1*
(iii) *if (J, v) is a representative with respect to R_1 and all the entries $R_1[j]$ for $j \in J$ are strictly smaller than $n + 1$ then (J, v) is also a representative with respect to R_2*

Proof. Simple extension of the proof of Lemma 1 in [19]. □

The following property is crucial for the validity of the pruning:

Lemma 2. *Let (J, v) be a representative under G with respect to rank vector R. Let $J' := J - j$ with j such that $R[j] = \max\{R[i] \mid i \in J\}$ and $v[j] = \min\{v[i] \mid i \in J \text{ with } R[i] = R[j]\}$ and let v' be the vector obtained from v by changing the value of $v[j]$ to -1. Then (J', v') is also a representative with respect to R.*

Proof. Simple extension of the proof of Lemma 2 in [19]. □

Consider the following *isomorphism pruning* to be applied at node a of the enumeration tree of a branch-and-cut using a ranked branching rule: If (F_p^a, f_p^a) is not a representative with respect to R^a then prune node a. (Node a is said to be *pruned by isomorphism* for short.)

Let $\mathcal{B}$ be a branch-and-cut using a ranked branching rule, isomorphism pruning, and a particular enumeration strategy. Let $\mathcal{T}$ be the enumeration tree of $\mathcal{B}$, assuming that nodes are pruned only by isomorphism pruning or when the LP relaxation of the corresponding ILP is infeasible. This implies that even in the case where the linear relaxation associated with node a has an integer optimal solution, $\mathcal{B}$ continues to branch. Pruned nodes are not included in $\mathcal{T}$.

Let $\mathcal{B}'$ be the branch-and-cut obtained from $\mathcal{B}$ by dropping isomorphism pruning, but enumerating the nodes in the same order as $\mathcal{B}$, the remaining nodes being processed arbitrarily after that. Let $\mathcal{T}'$ be the enumeration tree of $\mathcal{B}'$, assuming that nodes are pruned only by infeasibility. Pruned nodes are not included in $\mathcal{T}'$. Note that $\mathcal{T} \subseteq \mathcal{T}'$.

Lemma 3. *We have*

(i) *If $a \in \mathcal{T}' - \mathcal{T}$ then (F_p^a, f_p^a) is not a representative of its orbit;*
(ii) *$\mathcal{B}$ and $\mathcal{B}'$ return the same optimal value.*

Proof. Simple extension of the proof of Lemma 3 in [19]. □

Lemma 3 shows the validity of the isomorphism pruning. It should then be obvious that usual techniques such as cutting planes and pruning by bounds can be added to $\mathcal{B}$, keeping a branch-and-cut returning an optimal solution of the problem. However, some care should be taken when setting variables as explained in the next section.

4. ILP with general integer variables

In this section, we look at an ILP of the form ILP (1). The goal is to generalize the algorithms of [18,19] to the non binary case. This section is subdivided into five subsections: Section 4.1 is focused on methods to assign or exclude some values to a variable. It contains the main result of this paper, Lemma 6. Section 4.2 gives pointers to basic group algorithms and data structures. Sections 4.3 and 4.4 cover modifications to the algorithms of [19] for computing orbits and representatives in a group that are essential pieces in the branch-and-cut algorithm. Finally, Section 4.5 presents a comparison between several formulations for ILP with general integer variables. The example problem is the construction of orthogonal arrays, a classical problem in combinatorial design theory having a natural ILP formulation with variables that can take small integer values.

4.1. Setting variables

Using additional tools to set variables at a node a of the enumeration tree is possible, but some care should be taken in order to avoid conflicts with the isomorphism pruning. Let ILP^a denote the ILP at node a, i.e., ILP (1) where variables in $F^a \cup S^a$ are restricted to their respective values at a.

We consider a branch-and-cut $\mathcal{B}$ using isomorphism pruning and a ranked branching rule. Let $\mathcal{T}$ be the nodes in the enumeration tree of $\mathcal{B}$ that are not pruned by infeasibility or isomorphism. For $a \in \mathcal{T}$, let $\mathcal{T}^a$ be the subtree of $\mathcal{T}$ rooted at a. A *feasible leaf* of $\mathcal{T}^a$ is a leaf of $\mathcal{T}^a$ where all variables are fixed to a value in K. A *solution* in $\mathcal{T}^a$ is a solution x corresponding to a feasible leaf of $\mathcal{T}^a$. An *optimal solution* in $\mathcal{T}^a$ is a solution in $\mathcal{T}^a$ that is optimal for ILP (1).

The *strict setting algorithms* introduced in [19] can be generalized to the non binary case as follows: In the binary case, setting variable x_j to 0 or excluding the value 1 for x_j are equivalent. In the non binary case, however, it is much more likely that we are able to exclude some of the possible values for x_j than to set x_j to a specific value. A *strict setting algorithm* is a procedure that proves that, for some $j \in N$ and some $q \in K$, in any optimal solution $\bar{x}$ in $\mathcal{T}^a$, we have $\bar{x}_j \neq q$. The difference from the usual algorithms for excluding values for variables in a subproblem is that the usual algorithms only require proof that at least one optimal solution $\bar{x}$ in $\mathcal{T}^a$ satisfies $\bar{x}_j \neq q$.

An additional crucial property of a strict setting algorithm that can be used in conjunction with isomorphism pruning as described here is that it works under symmetry: If the setting algorithm is able to show that x_j can not take value q in ILP^a then, for any $g \in G$, it is able to reach the same conclusion for variable $x_{g(j)}$ in the ILP obtained from ILP^a by permuting the variables according to g. This essentially prevents the setting algorithm to work based on conditions linked to the isomorphism pruning, but allows for traditional strict setting procedures (for example, strict versions of strong branching or reduced cost fixing [19]). In the remainder of the paper, we only consider strict setting algorithms working under symmetry.

The wider choice of possible values for x_j compared to the binary case requires more bookkeeping. The information required by the pruning algorithm is the point where it became possible to show that x_j can not have value q in ILP^a. This information is stored in an $n \times (k+1)$ *date matrix* denoted by D^a, with $D[j,q] = t$ if, at an ancestor node b with $|F_p^b| = t$, we could conclude that x_j can not take value q in any optimal solution in $\mathcal{T}^b$. Initially, the date matrix of the root node is filled with entries with value $n+1$. When dealing with a node a that is not the root node, the matrix D^a is initially a copy of D^b, where b is the parent node of a in the tree. Then, if the branching decision creating a from b is to fix variable $x_h = q$, index h is included in F_p^a or in F_0^a and D^a is modified as follows:

$$D^a[h,q'] = \begin{cases} |F_p^b| & \text{if } q' > q, \\ |F_p^a| & \text{if } q' < q. \end{cases} \tag{2}$$

The distinction between $q' > q$ and $q' < q$ is natural, as any valset (J, v) corresponding to $F_p^b \cup h$ with $x_h = q' > q > 0$ is lexicographically smaller than F_p^a. This valset as well as any valset in its orbit under G should not appear in $\mathcal{T}^a$ due the definition of the isomorphism pruning. On the other hand, if (J, v) corresponds to to $F_p^b \cup h$ with $x_h = q' < q$ and $q' > 0$, it is lexicographically larger than F_p^a. This valset can not appear in $\mathcal{T}^a$ because $x_h = q$ in $\mathcal{T}^a$, but a valset in its orbit is allowed to exist in $\mathcal{T}^a$.

Note that when $D^a[j,q]$ is said to be set to value t or when we write, as above, that $D^a[j,q] = t$ it should be understood that the entry is modified only if its current value is larger than t.

If, during the processing of node a, it can be shown that x_j can not take value q, then $D^a[j,q]$ is set to $|F_p^a|$. Thus, at node a, either $D^a[j,q] \leq |F_p^a|$ or $D^a[j,q] = n+1$. The possible values q for x_j in ILP^a correspond to the entries $D^a[j,q] = n+1$. Hence, if exactly one entry in row j of D^a has value $n+1$, then $j \in F^a \cup S^a$ and x_j is fixed or set to the corresponding value q. Note also that if, at any time, there is $j \in N$ such that all entries in row j of D^a are smaller than $n+1$, then ILP^a is infeasible and node a can be pruned. This is implicit in all the algorithms presented below.

Besides a strict setting algorithm, we also use the following observation for excluding some values for variables in ILP^a: Let (J, v) be a valset with $v[j] > 0$ for all $j \in J$ and let F be the subset of F_p^a containing its $|J|$ indices with smallest rank with respect to R^a (take $F = F_p^a$ if $|J| \geq |F_p^a|$). Let f be the n-vector corresponding to the values of the variables in F. Suppose that $(g(J), g(v)) \prec (F, f)$ for some $g \in G$. Then, for at least one index $j \in J$, we must have $x_j < v[j]$. This is true as, in $\mathcal{T}^a$, all feasible leaves b have that (F_p^b, f_p^b) is a representative and, as $F \subseteq F_p^b$, the valset (J, v) can not have a representative lexicographically smaller than (F, f) due to isomorphism pruning. The next lemmas exploit this idea on a subset for which at most one variable is not fixed or set to a particular value.

Lemma 4. *Let $a \in \mathcal{T}$, let $J \subseteq F_p^a \cup S_p^a$ with $|J| < |F_p^a|$ and $s \notin J, q > 0$ with $D^a[s,q] = n+1$. Let v be the n-vector with*

$$v[j] = \begin{cases} fs_p^a[j] & \text{for all } j \in J, \\ q & \text{for } j = s, \\ -1 & \text{otherwise.} \end{cases}$$

Let F be the subset of F_p^a containing its $|J|+1$ indices with smallest rank with respect to R^a. Let f be the n-vector with $f[j] = f_p^a[j]$ for all $j \in F$ and $f[j] = -1$ otherwise. Suppose that there exists $g \in G$ with $(g(J \cup s), g(v)) \prec (F, f)$. Then it is valid to set $D^a[s,q] = |F_p^a|$.

Proof. Let $\bar{x}$ be any optimal solution in $\mathcal{T}^a$. If $\bar{x}_s = q$ then $(J \cup s, v)$ is a valset corresponding to a subset of the variables having positive values in $\bar{x}$. This valset has a representative strictly smaller than (F, f), a contradiction with isomorphism pruning. □

Example 1. Let us illustrate Lemma 4 on the following small example: Assume that the ILP has seven integer variables bounded between 0 and 2 and that G is the group generated by the permutations $[2,3,4,5,6,7,1]$ and $[7,6,5,4,3,2,1]$. (This group is the symmetry group of a regular polygon with 7 vertices, i.e. the dihedral group on seven elements.) Assume that the minimum index branching rule is used and that, at node a, we have the following variables fixed by branching: $x_1 = 2, x_2 = 1$. We thus have $F_p^a = \{1,2\}$ and $S_p^a = \emptyset$. A date matrix consistent with the above is:

$$D^a = \begin{bmatrix} 1 & 2 & 8 & 8 & 8 & 8 & 8 \\ 1 & \mathbf{8} & 8 & 8 & 8 & 8 & 8 \\ \mathbf{8} & 1 & 8 & 8 & 8 & 8 & 8 \end{bmatrix}^{\mathrm{T}}.$$

(Note that to save space the transpose of D^a is displayed, that the first row in the display has index 0 and that the first column has index 1; boldface is used to indicate entries corresponding to fixed or set variables.) Let $J = \{1\}, s = 7$ and $q = 2$. Then $v = [2,-1,-1,-1,-1,-1,2]^{\mathrm{T}}$, $F = \{1,2\}$ and $f = [2,1,-1,-1,-1,-1,-1]^{\mathrm{T}}$. For $g = [1,7,6,5,4,3,2]$, we have $(g(J \cup s), g(v)) \prec (F, f)$. By Lemma 4, it is thus valid to set $D^a[7,2] = 2$, excluding value 2 for variable x_7 in the subtree $\mathcal{T}^a$. □

Lemma 5. *Let $a \in \mathcal{T}$, let $J \subseteq F_p^a \cup S_p^a$ with $|J| \leq |F_p^a|$ and $s \notin J, q > 0$ with $D^a[s,q] = n+1$. Let v be the n-vector with*

$$v[j] = \begin{cases} fs_p^a[j] & \text{for all } j \in J, \\ -1 & \text{otherwise.} \end{cases}$$

Let F be the subset of F_p^a containing its $|J|$ indices with smallest rank with respect to R^a. Let f be the n-vector with $f[j] = f_p^a[j]$ for all $j \in F$ and $f[j] = -1$ otherwise. Let

$$\bar{r} = \begin{cases} \min\{R^a[j] \mid j \in F_p^a - F\} & \text{if } F_p^a \neq F, \\ \max\{R^a[j]+1 \mid j \in F_0^a\} & \text{if } F_p^a = F \text{ and } F_0^a \neq \emptyset, \\ 0 & \text{otherwise.} \end{cases}$$

Suppose that there exists $g \in G$ with $(g(J), g(v)) = (F, f)$ and $R^a[g(s)] < \bar{r}$. Then it is valid to set $D^a[s, q] = |F_p^a|$.

Proof. Assume that there exists an optimal solution $\bar{x}$ in $\mathcal{T}^a$ such that $\bar{x}_s > 0$. Let $i \notin F$ with minimum rank and such that $\bar{x}_i > 0$. Then $R^a(g(s)) < \bar{r} \leq R^a(i) \leq R^a(s)$, as $x_{g(s)}$ is fixed to 0 before i is fixed to a positive value. But then the valset (F^*, v^*) corresponding to the positive entries in $\bar{x}$ is not lexicographically minimum in its orbit under G, as demonstrated by applying g to the valset corresponding to the entries in $(J \cup s)$ in $\bar{x}$, a contradiction with isomorphism pruning. □

Example 2. Using the same group and ILP as in Example 1, an illustration of Lemma 5 is the following: Assume that at node a we have the following variables fixed by branching: $x_1 = 2, x_2 = 1, x_3 = 0$. Observe that application of Lemma 4 after the first two variables are fixed yields that value 2 is excluded for x_7 (as seen in Example 1). Suppose that using a strict setting algorithm after the first three variables are fixed shows that x_7 can not have value 0 in any optimal solution in $\mathcal{T}^a$. Then, x_7 is set to 1. We thus have $F_p^a = \{1, 2\}$, $S_p^a = \{7\}$, $F_0^a = \{3\}$, and a date matrix consistent with the above is:

$$D^a = \begin{bmatrix} 1 & 2 & \mathbf{8} & 8 & 8 & 8 & 3 \\ 1 & \mathbf{8} & 2 & 8 & 8 & 8 & \mathbf{8} \\ \mathbf{8} & 1 & 2 & 8 & 8 & 8 & 2 \end{bmatrix}^{\mathrm{T}}.$$

It is then possible to apply Lemma 5 with $J = \{1, 7\}, s = 6, \bar{r} = 4$ and $q = 1, 2$. Then $v = [2, -1, -1, -1, -1, -1, 1]^{\mathrm{T}}$, F, and f are as in Example 1 and for $g = [1, 7, 6, 5, 4, 3, 2]$, we get that we can set $D^a[6, 2] = 3$ and $D^a[6, 1] = 3$, thus setting $x_6 = 0$ in the subtree $\mathcal{T}^a$. □

The next lemma shows that a more general result holds. Corollaries of Lemma 6 will indeed show that Lemmas 4 and 5 are weaker.

Lemma 6. *Let $a \in \mathcal{T}$, $J \subseteq F_p^a \cup S_p^a$ with $|J| \leq |F_p^a|$, $s \notin J$, $q > 0$ such that $D^a[s, q] = n + 1$, and $g \in G$. Let F be the subset of F_p^a containing its $|J|$ indices with smallest rank with respect to R^a. Let f be the n-vector with $f[j] = f_p^a[j]$ for all $j \in F$ and $f[j] = -1$ otherwise. Let $\bar{s} \in J \cup s$ and let the n-vector v with*

$$v[j] = \begin{cases} fs_p^a[j] & \text{for all } j \in J - \bar{s}, \\ q & \text{if } j = s \text{ and } s \neq \bar{s}, \\ -1 & \text{otherwise} \end{cases}$$

and

$$\bar{q} = \begin{cases} fs_p^a[\bar{s}] & \text{if } \bar{s} \neq s, \\ q & \text{otherwise.} \end{cases}$$

If we have $(g((J \cup s) - \bar{s}), g(v)) = (F, f)$ and $D^a[g(\bar{s}), \bar{q}] \leq |J|$ then it is valid to set $D^a[s, q] = |F_p^a|$.

Example 3. Using the same group and ILP as in Example 1, an illustration of Lemma 6 is the following: Assume that at node a we have the following variables fixed by branching: $x_1 = 2, x_2 = 1, x_3 = 2, x_4 = 1$. As seen in Example 1, value 2 can be excluded for x_7 after the first two variables are fixed. Assume now that value 1 for x_6 and value 0 for x_7 can be excluded by a strict setting algorithm just after x_3 is fixed to 2. We thus have $F_p^a = \{1, 2, 3, 4\}$ and $S_p^a = \{7\}$. A date matrix consistent with the above is:

$$D^a = \begin{bmatrix} 1 & 2 & 3 & 4 & 8 & 8 & 3 \\ 1 & \mathbf{8} & 3 & \mathbf{8} & 8 & 3 & \mathbf{8} \\ \mathbf{8} & 1 & \mathbf{8} & 3 & 8 & 8 & 2 \end{bmatrix}^{\mathrm{T}}.$$

Let $J = \{3, 4, 7\}, s = 5, q = 2, \bar{s} = 7$. Then $v = [-1, -1, 2, 1, 2, -1, -1]^{\mathrm{T}}$ and $\bar{q} = 1$. Let $F = \{1, 2, 3\}$ and $f = [2, 1, 2, -1, -1, -1, -1]^{\mathrm{T}}$. For $g = [5, 4, 3, 2, 1, 7, 6]$, we have $(g((J \cup s) - \bar{s}), g(v)) = (F, f)$ and $D^a[g(\bar{s}), \bar{q}] = D^a[6, 1] = 3 \leq |J| = 3$. By Lemma 6, it is thus valid to set $D^a[5, 2] = 4$, excluding value 2 for variable x_5 in the subtree $\mathcal{T}^a$.

Proof (*Lemma* 6). The result holds unless there exists an optimal solution $\bar{x}$ in $\mathcal{T}^a$ with $\bar{x}_s = q$. We will show that if we assume that such an optimal solution exists then, since we could prove that $\bar{x}_{g(\bar{s})} \neq \bar{q}$, we have that $\bar{x}_{\bar{s}} \neq \bar{q}$. If $s = \bar{s}$, this yields a direct contradiction with $\bar{x}_s = q$. If $s \neq \bar{s}$, then $\bar{s} \in F_p^a \cup S_p^a$, and $\bar{x}_{\bar{s}} = \bar{q}$, a contradiction too.

Consider the sequence of fixing and exclusion of values that occurred along the path from the root to the node (a or one of its ancestors) where it was proved that $x_{g(\bar{s})} \neq \bar{q}$. We can encode these operations as $(i_1, q_1), \ldots, (i_t, q_t)$ with the meaning that it was proved that $\bar{x}_{i_u} \neq q_u$ for $u = 1, \ldots, t$ in that order, with $(i_t, q_t) = (g(\bar{s}), \bar{q})$. Observe that a pair (i_u, q_u) appears at most once in this sequence. This allows us to assume without loss of generality that the settings obtained using Lemma 6 obey the following rule: If a setting using Lemma 6 is possible, it is done. Moreover, if two such settings may be done, the one done first is obtained using the smallest u such that entry $D^a[i_u, q_u]$ is the entry that is $\leq |J|$ in the statement of the lemma. We also assume that s and q are chosen such that all settings made before setting $D^a[s, q] = |F_p^a|$ were valid for all optimal solutions in $\mathcal{T}^a$.

Let $\text{val}(i_u, q_u)$ be the value used to set $D^a[i_u, q_u]$ when the exclusion is done. We assume without loss of generality that $\text{val}(i_u, q_u) \leq \text{val}(i_{u+1}, q_{u+1})$ for all $u = 1, \ldots, t-1$. This is indeed trivially met for all setting operations since, for a setting operation at node b, $\text{val}(i_u, q_u) = |F_p^b|$. For fixing operations, this will hold if we assume that fixing x_h to q' is done by first excluding all values $q > q'$ and then excluding the values $q < q'$, as can be seen from (2). As a result, and since $D^a[g(\bar{s}), \bar{q}] \leq |J|$, we have that

$$D^a[i_u, q_u] \leq |J| \quad \text{for all } u = 1, \ldots, t. \tag{3}$$

We prove by induction on u that if we know that $\bar{x}_{g^{-1}(i_j)} \neq q_j$ for $j = 1, \ldots, u-1$ then $\bar{x}_{g^{-1}(i_u)} \neq q_u$. We will use several times the fact that, by the induction hypothesis, all excluded values for i when (i_u, q_u) is obtained are also excluded for $g^{-1}(i)$, for all $i \in N$. This fact will be referenced by the term *exclusion symmetry* for short.

- If (i_u, q_u) was obtained by using a strict setting algorithm, the fact that the algorithm works under symmetry allows us to exclude $\bar{x}_{g^{-1}(i_u)} = q_u$ using the exclusion symmetry. Hence, we must have $\bar{x}_{g^{-1}(i_u)} \neq q_u$.
- If (i_u, q_u) was obtained by application of Lemma 6, let us use a superscript * for all symbols corresponding to that application of Lemma 6: It uses a set J^*, indices $s^* = i_u$ and $\bar{s}^*$, values $q^* = q_u$ and $\bar{q}^*$, and a permutation g^*. By exclusion symmetry, we can use Lemma 6 with set $g^{-1}(J^*)$, indices $g^{-1}(i_u)$ and $g^{-1}(\bar{s}^*)$, values q_u and $\bar{q}^*$, and permutation $(g^* \cdot g)$ and obtain that $g^{-1}(i_u)$ can not have value q_u. Let u' be such that $(i_{u'}, q_{u'}) = (g^*(i_u), \bar{q}^*)$. The assumption on the ordering of the setting operations and the choice of s, q imply that $u' < u$ and that $(g^{-1}(i_u), q_u)$ is done before we try to set $D^a[s, q] = |F_p^a|$ and thus is valid. Hence, we must have $\bar{x}_{g^{-1}(i_u)} \neq q_u$.
- If (i_u, q_u) was obtained by branching on x_{i_u} and fixing it to a value $q' > 0$ then one of the two following cases occur:
  (a) $q_u < q'$. The date setting update rule (2) and inequality (3) imply that i_u is one of the first $|J|$ variables fixed to a positive value. It follows that $i_u \in F$ since F contains the first $|J|$ variables fixed to a positive value. As $(g((J \cup s) - \bar{s}), g(v)) = (F, f)$, we have either that $g^{-1}(i_u) \in F_p^a \cup S_p^a$ and is already fixed or set to q' in ILP^a, or $g^{-1}(i_u) = s$. In the latter case, we have $q' = q$ and $\bar{x}_s = q' \neq q_u$ as requested.
  (b) $q_u > q'$. Then either $i_u \in F$ and the above reasoning applies, or i_u is the $(|J|+1)$-th variable fixed to a positive value. Observe that Lemma 4 then yields the result: Using a superscript * for all symbols corresponding to those in Lemma 4, take $J^* := (J \cup s) - \bar{s}$, $s^* := g^{-1}(i_u)$, $q^* := q$, and $g^* := g$.
- If (i_u, q_u) was obtained by branching on x_{i_u} and fixing it to 0, then the date setting update rule (2) and inequality (3) imply that at most $|J|$ variables are already fixed to a positive value. Let b be the ancestor of a obtained after the fixing $x_{i_u} = 0$. Assume that $\bar{x}_{g^{-1}(i_u)} = q' > 0$ and observe that $F_p^b \subseteq F$. One of the two following cases occur:
  (a) If $s = \bar{s}$, then $g^{-1}(F_p^b) \subseteq J$ and, as $(g(J), g(v)) = (F, f)$, all variables in $g^{-1}(F_p^b)$ are fixed or set to similar values in $\bar{x}$. Lemma 5 then yields a contradiction with $\bar{x}_{g^{-1}(i_u)} = q' > 0$: Using a superscript * for all symbols corresponding to those in Lemma 5, take $J^* := g^{-1}(F_p^b)$, $s^* := g^{-1}(i_u)$, $q^* := q'$, and $g^* := g$.
  (b) If $s \neq \bar{s}$, let d be the feasible leaf in $\mathcal{T}^a$ corresponding to $\bar{x}$. As $\bar{s} \in F_p^a \cup S_p^a$, F_p^d contains at least one index more than F_p^b. Let $j \in F_p^d - F_p^b$ with minimum rank with respect to R^a. Let $J' = g^{-1}(F^b \cup i_u)$ and v' be the n-vector with $v'[j] = \bar{x}[j]$ for all $j \in J'$ and -1 otherwise. Observe that $(g(J'), g(v'))$ is lexicographically smaller than (F_p^d, f_p^d), since $R^a(i_u) < R^a(j)$. This yields a contradiction, since d is not pruned by isomorphism. □

One very attractive property of Lemma 6 is that no assumption is made on previous setting operations. It is sometimes possible to get similar results only if one assumes that all possible settings are done at each stage. This is not the case here, as only the validity of previous settings is required.

The next two corollaries prove that Lemma 6 supersedes earlier results.

Corollary 1. *Any setting done using Lemma 4 can be done using Lemma 6.*

Proof. Using a superscript * for all symbols corresponding to those in Lemma 4 yielding $D^a[s^*, q^*] = |F_p^a|$, take $J := J^*$, $s := s^*$, $q := q^*$, and $g := g^*$ in Lemma 6 and $\bar{s} = (J \cup s) - g^{-1}(F)$. Let $\bar{f}$ be the $|J^*| + 1$-th variable fixed to a positive value. Then, $(g^*(J^* \cup s^*), g^*(v^*)) \prec (F^*, f^*)$ implies that one of the following holds:

- $R[g(\bar{s})] < R[\bar{f}]$. Then $g(\bar{s}) \in F_0^a$.
- $R[g(\bar{s})] = R[\bar{f}]$ and $\bar{q} > f_p^a[\bar{f}]$. The first part of the condition implies $g(\bar{s}) = \bar{f}$.

In both cases, the update (2) of the date matrix when branching on variable $g(\bar{s})$ implies that $D^a[g(\bar{s}), \bar{q}] \leq |J|$ and Lemma 6 applies. □

Corollary 2. *Any setting done using Lemma 5 can be done using Lemma 6.*

Proof. Using a superscript * for all symbols corresponding to those in Lemma 5 yielding $D^a[s^*, q^*] = |F_p^a|$, take $J := J^*$, $s := s^*$, $q := q^*$, $g := g^*$ and $\bar{s} = s$ in Lemma 6. Since $R[g(s^*)] < \bar{r}^*$, we have that $g(s^*) \in F_0^a$, implying that $g(\bar{s}) \in F_0^a$. The update (2) of the date matrix when branching on variable $g(\bar{s})$ implies that $D^a[g(\bar{s}), \bar{q}] \leq |J|$ and Lemma 6 applies. □

The difficulty in using Lemma 6 is the large number of sets J to consider, and, even for a given J, deciding if there exists a $g \in G$ as specified in the statement of the lemma is difficult. To get practical implementations based on the lemma, the following corollary is useful:

Corollary 3. *Let $a \in \mathcal{T}$ and let $s \notin F_p^a$, and $q > 0$ with $D^a[s, q] = n + 1$. Let $\mathcal{O} = \text{orb}(s, \text{stab}(F_p^a, f_p^a, G))$. If there exists $s' \in \mathcal{O}$ with $D^a[s', q] \leq |F_p^a|$ then it is valid to set $D^a[s'', q] = |F_p^a|$ for all $s'' \in \mathcal{O}$.*

Proof. This comes directly from Lemma 6: Using a superscript * for all symbols corresponding to those in Lemma 6 take $J^* := F_p^a$, $s^* := s''$ and $\bar{s}^* := s''$. Since both s' and s'' are in $\mathcal{O}$, there exists $g \in G$ with $g(F_p^a, f_p^a) = (F_p^a, f_p^a)$ and $g(s'') = s'$. □

Consider the following operations at node $a \in \mathcal{T}$ with rank vector R^a, called an *orbit setting*. Let *set_alg(a)* be the strict setting algorithm used at node a:

(i) Compute all orbits $\mathcal{O}_1, \ldots, \mathcal{O}_z$ in $\text{stab}(F_p^a, f_p^a, G)$.
(ii) For each $i \in \{1, \ldots, z\}$, for each $q \in K$, let $m_i(q) = \min\{D^a[j, q] \mid j \in \mathcal{O}_i\}$. For each $j \in \mathcal{O}_i$, for each $q > 0$ with $m_i(q) \leq |F_p^a|$, set $D^a[j, q] = |F_p^a|$. Update S_0^a and S_p^a accordingly.
(iii) If additional variables can be set by *set_alg(a)* update S_0^a and S_p^a accordingly and go to (ii).
(iv) If $N^a = \emptyset$ then return $n + 1$ and stop.
(v) Let x_h be the variable that would be chosen as the branching variable, according to the ranked branching rule. Return h and stop.

The output of the orbit setting is the index of the variable x_h that will be branched on, or $n + 1$ if no such h exists. The validity of the orbit setting should be clear. Step (ii) is an application of Corollary 3. It remains to show how to compute orbits in $\text{stab}(F_p^a, f_p^a, G)$ and how to test if a valset is a representative or not. This will be covered in the next three subsections. If orbit setting is used, the operations performed at node a are:

$h :=$ orbit setting at a;
Repeat until a criterion is met
 solve the LP relaxation of ILP^a;
 generate cuts;
If $h < n + 1$ then create the son d_q of a by fixing $x_h = q$ for each q with $D^a[h, q] = n + 1$.
Update R, set up $F_p^{d_q}$ and update D^{d_q} according to (2).

Let $\mathcal{O}$ be the orbit of h in stab(F_p^a, f_p^a, G). In the son d_q, set $D^{d_q}[j, \bar{q}] = |F_p^a|$ for all $j \in \mathcal{O}$, for all $\bar{q} > q$.

The validity of the setting made in the last paragraph above is Lemma 4 applied at node d_q with $J = F_p^a$ and $s \in \mathcal{O}$.

Note that computing exactly the orbits in point (i) of the orbit setting is not essential. It is possible to use a partial orbit instead of a full orbit or to have orbits broken into several pieces. The orbit setting will be weaker, but remains valid. Since computing completely all the orbits in the stabilizer of a valset might be time consuming, the implementation of the orbit setting in the code used in the tests only computes some of the orbits, and some not always completely. More precisely, when testing if a valset $(F_p^a \cup h, v)$ is a representative, the process is stopped as soon as a permutation is found proving that the valset is not a representative. At that point, usually, only part of the orbit of h is known. This partial orbit is used in our implementation of the orbit setting. On the other hand, if the valset is a representative, then the complete orbit of h in stab(F_p^a, f_p^a, G) is computed.

4.2. Group operations

The chosen group representation and algorithms are based on the *Schreier–Sims* representation of G [2–5,11,12, 14–16]. The reader is referred to the papers [18,19] for a more detailed presentation.

Let $G_0 = G$ and $G_i = \text{stab}(i, G_{i-1})$ for $i = 1, \ldots, n$. Observe that $G_0, G_1, \ldots, G_n$ are nested subgroups of G.

For $t = 1, \ldots, n$, let $\text{orb}(t, G_{t-1}) = \{j_1, \ldots, j_p\}$ be the orbit of t under G_{t-1}. Then for each $1 \le i \le p$, let h_{t,j_i} be any permutation in G_{t-1} sending t on j_i, i.e., $h_{t,j_i}[t] = j_i$. Let $U_t = \{h_{t,j_1}, \ldots, h_{t,j_p}\}$. Note that U_t is never empty as $\text{orb}(t, G_{t-1})$ always contains t.

Arrange the permutations in the sets $U_t, t = 1, \ldots, n$ in an $n \times n$ table T, with

$$T_{t,j} = \begin{cases} h_{t,j} & \text{if } j \in \text{orb}(t, G_{t-1}), \\ \emptyset & \text{otherwise.} \end{cases}$$

The table T is called the Schreier–Sims representation of G. It is possible to make a small generalization of the presentation by ordering the points of the ground set in an arbitrary order β, called the *base* of the table. In that case, the subgroups $G(\beta)_t$ for $t = 1, \ldots, n$ are defined as the stabilizer of $\beta[t]$ in $G(\beta)_{t-1}$, with $G(\beta)_0 = G$. The corresponding table is denoted by $T(\beta)$. Row t of $T(\beta)$ corresponds to the element t, $U(\beta)_t$ is the set of non empty entries in row t of $T(\beta)$ and $J(\beta)_t$ denotes the corresponding set of indices $\{j \in I^n \mid T(\beta)[t, j] \neq \emptyset\}$, also called the *basic orbit* of t in T (following the terminology of [16]). When the base β is fixed, we sometimes drop the qualifier (β) in these symbols, but from now on each table T is defined with respect to a base.

Although the algorithms are described for a 2-dimensional table T, a more space efficient implementation uses a vector of ordered lists instead, as most entries in the table are usually empty. (See [18] for more details regarding non empty entries in the table.) The actual implementation uses a vector of ordered lists, but algorithms are simpler to describe and understand for the 2-dimensional table.

We use backtracking algorithms to decide if a set is a representative or to compute the orbits in the stabilizer of a set in G. These algorithms take advantage of the fact that we may assume that the base β of the group at node a of the enumeration tree has the following structure: Variables fixed to positive values at a (i.e., F_p^a) come first in β, then the variables not set to 0 and not fixed (i.e. $N^a - S_0^a$), and then the variables fixed or set to 0 at a (i.e. $F_0^a \cup S_0^a$).

The data structure associated with group G at node a of the branch-and-cut is the following:

`integer:` bvf	`integer:` n_f
`integer vector:` f_p	`matrix of permutations:` T
`integer vector:` β	`matrix of integer:` D

In addition, a single rank vector R is updated during the whole enumeration according to the rule of Section 3. When processing node a, the current rank vector R corresponds to the vector R^a of the previous sections. The integer bvf is the index of the branching variable of the father of a. The variable n_f gives the number of variables in F_p^a and

$$F_p^a = \beta[1..n_f] \quad \text{with } R[\beta[1]] < \cdots < R[\beta[n_f]].$$

The vector f_p is the vector f_p^a of the previous section. All variables in $F_0^a \cup S_0^a$ are moved at the end of the base at the time they are fixed or set to 0. The remaining variables appear in β in increasing order of their rank, after variables in F_p^a and before variables in $F_0^a \cup S_0^a$. This structure of β is not difficult to maintain throughout the branch-and-cut,

using the procedure *down*() of [18] and a more general base change algorithm when needed. The procedure *down*() has complexity $O(n^6)$ for downing a point. The table T is just a Schreier–Sims representation of the group with base β. The matrix D is the date matrix D^a of the previous subsection.

In this paper, we consider algorithms for solving questions related to a single node a of the branch-and-cut. To avoid heavy notations, the table associated with a is denoted by T, instead of a more precise notation like $T(a)$ or $a \to T$. The same remark applies to the other fields of the data structure associated with a.

We are interested in performing the following operations that were mentioned in Section 4.1: Computing all orbits in the stabilizer of a valset and deciding if a valset is lexicographically minimum in its orbit under G.

4.3. Computing orbits in the stabilizer of a valset

One property of a Schreier–Sims representation of G is that each $g \in G$ can be uniquely written as

$$g = g_1 \cdot g_2 \cdot \cdots \cdot g_n \tag{4}$$

with $g_i \in U(\beta)_i$ for $i = 1, \ldots, n$. Hence the permutations in the table form a set of generators of G. As a consequence, any $g \in G$ can be written as

$$g = g_1 \cdot \cdots \cdot g_{t-1} \cdot g_t \cdot h$$

with $g_i \in U(\beta)_i$ for $i = 1, \ldots, t$, and $h \in G_t$.

The backtracking procedure given below outputs generators of the stabilizer of the valset $(\beta[1..t], v)$, where the vector v is such that $v[\beta[j]]$ is the value associated with $\beta[j]$ for $j = 1, \ldots, t$ and -1 otherwise. It consists of an initializing procedure stabilizer_gen() that calls a recursive procedure stab_gen(). This is a very slight modification of the corresponding procedure in [19]. The proof is not repeated here, as the extension to valsets is straightforward.

```
stabilizer_gen(a, v, t)

/* Outputs generators of stab(β[1..t], v, G) where G is the group represented
by T with base β */

  Output U(β)_i for i = t + 1, ..., n;
  ident = identity permutation;
  remain := β[1..t];
  stab_gen(a, v, t, ident, remain, 1);
```

The parameters of the call to stab_gen() have the following interpretation: *ind* refers to the point $\beta[ind]$ being treated during the current call; *perm* is a permutation in G sending $\beta[1..ind-1]$ on a subset $B \subseteq \beta[1..n_f]$ with $v[\beta[i]] = v[perm(\beta[i])]$ for $i = 1, \ldots, ind-1$ and *remain* is the set $perm^{-1}(\beta[1..n_f] - B)$.

```
stab_gen(a, v, t, perm, remain, ind)

  For each i ∈ remain do
    h := T[β[ind], i];
    If h ≠ ∅ then
      If f_p[perm(i)] = f_p[β[ind]] then
        loc_remain := remain − i;
        loc_remain := h^{-1}(loc_remain);
        loc_perm := perm · h;
        If ind < t then
          stab_gen(a, v, t, loc_perm, loc_remain, ind + 1);
        else
          output perm.
```

Remark 1. Observe that if we run the algorithm for $t = |F_p^a|$ then, at the beginning of a recursive call at depth *ind*, it is possible to use Lemma 6 to possibly reduce some entries in the date matrix D^a: In the notation of the lemma, we can use $J := perm(\beta[1..ind-1])$, $g := perm^{-1}$, $s \notin J$ and $\bar{s} := s$. Then if $D^a[g(s), q] \leq ind - 1$ then we can set $D^a[s, q] = |F_p^a|$. This is implemented in the code tested in Section 4.5. □

4.4. Deciding if a valset is a representative or not

For deciding if a set is a representative of its orbit with respect to R, we refer to [18], where procedure *first_in_orbit*() is described. The complexity of the procedure is $O(n \cdot t!)$, where t is the cardinality of the set.

Here, we are only interested in valsets that are representative of their orbit under G. The following procedure tests if the valset $(\beta[1..n_f + 1], v)$ is a representative, where v is obtained from a copy of f_p and by setting $v[\beta[n_f + 1]] = q > 0$. This is a very slight modification of the corresponding procedure in [19].

```
first_in_orbit(a, v)

/* Returns "true" if and only if (β[1..n_f + 1], v) is a
representative. */

   ident := identity permutation;
   remain := β[1..n_f + 1];
   is_lexmin := true;
   f_in_orb(a, v, ident, remain, 1, is_lexmin);
   return(is_lexmin);
```

The Boolean parameter *is_lexmin* is passed by reference. As soon as *is_lexmin* = `false`, it is known that $(\beta[1..n_f + 1], v)$ is not a representative and the procedure stops.

```
f_in_orb(a, v, perm, remain, ind, is_lexmin)

   If is_lexmin = false then return;
   For each i ∈ remain do
        If R[i] < R[β[ind]] then                    (*)
          is_lexmin := false;
          return;
   For each i ∈ remain do
        h := T[β[ind], i];
        If h ≠ ∅ then
          If v[perm[i]] > v[β[ind]] then             (**)
            is_lexmin := false;
            return;
          If v[perm[i]] = v[β[ind]] then
            loc_remain := remain − i;
            loc_remain := h^{-1}(loc_remain);
            loc_perm := perm · h;
            If ind < n_f then
              f_in_orb(a, v, loc_perm, loc_remain, ind + 1,
                                                    is_lexmin);
```

Lemma 7. *The algorithm first_in_orbit() is correct.*

Proof. Observe that at depth *ind*, if we define $B = perm(\beta[1..ind-1])$, then $B \subseteq \beta[1..n_f + 1]$ and $perm(remain) = \beta[1..n_f + 1] - B$.

Suppose that the algorithm returns `false`. Then, during a call at depth *ind* where (*) is satisfied, applying the permutation $perm^{-1}$ to the valset corresponding to indices in $B \cup perm(i)$ proves that the valset is indeed not a representative. Similarly, during a call at depth *ind* where (**) is satisfied, applying the permutation $(h^{-1} \cdot perm^{-1})$ to the valset corresponding to indices in $B \cup perm[i]$ proves that the valset is indeed not a representative: as h fixes all the points in $\beta[1..ind-1]$, we have $(h^{-1} \cdot perm^{-1})(B \cup perm[i]) = \beta[1 \cdots ind]$ and (**) ensures that $v[perm[i]] > v[\beta[ind]]$.

If the valset is not a representative, there exists $J \subseteq \beta[1..n_f + 1]$ and $g \in G$ that proves it. If J is chosen minimal with respect to inclusion, $J = J' \cup j$ such that $g(J') = \beta[1 \cdots |J| - 1]$ and either we have $R[g(j)] < R[\beta[|J|]]$ or we have $R[g(j)] = R[\beta[|J|]]$ and $v[g[j]] > v[\beta[|J|]]$. Then g will be the permutation $perm^{-1}$ or $(h^{-1} \cdot perm^{-1})$ in a recursive call at depth $|J|$ and the algorithm will return `false`. □

Remark 2. It is possible to use Lemma 6 for modifying entries in D^a while running this algorithm: Just before the first block starting with "For each $i \in remain$ do", let $B = perm(\beta[1..ind-1])$ and $g := perm^{-1}$.

- If $\beta[n_{f+1}] \in B$ then take $s := \beta[n_{f+1}]$ and, for any $i \in remain$, let $\bar{s} := perm(i)$, $\bar{q} = f^a_p[\bar{s}]$, and $J := (B \cup \bar{s}) - s$. Then $g(\bar{s}) = i$ and thus we can set $D^a[s, q] = |F^a_p|$ if $D^a[i, \bar{q}] \leq |J| = ind - 1$ for some $i \in remain$.
- If $\beta[n_{f+1}] \notin B$ then take $s \notin B$, $\bar{s} := s$, $q > 0$, and $J := B$. We can set $D^a[s, q] = |F^a_p|$ if $D^a[g(s), q] \leq ind - 1$.

This is implemented in the code tested in Section 4.5. □

Remark 3. The similarities between *stabilizer_gen()* and *first_in_orbit()* allow for a merge of the two algorithms, but for clarity they are presented separately here. Note also that it is possible to modify *first_in_orbit()* in order to treat simultaneously all possible values for q. These two improvements are implemented in the code used in the tests below. □

4.5. Computational results

The code is based on the COIN-OR open-source codes BCP (Branch, Cut & Price) and Clp (an LP solver), which are freely available from [7]. The machine used is a Dell Precision 650 (Intel Xeon processor, 8KB level-1 cache) running RedHat Enterprise Linux 3 and the code is generated using the gcc compiler (version 3.2).

All the reported results are on ILP where the value of the initial LP relaxation is the optimal ILP value or on ILP with known optimal value $\hat{z}$. When the problem is not in the former family, the branch-and-cut is used to prove that no solution with value better than $\hat{z}$ exists, i.e., $\hat{z}$ is used to prune the enumeration tree from the start. This is done in order to remove the randomness of the time at which an optimal solution is found. Since the optimal value $\hat{z}$ is always an integer for the problems under consideration, the value $\hat{z} - 0.99$ is used as the upper cutoff value. The branching variable order is the minimum index branching variable described in Section 3. The actual implementation of the branching rule takes advantage of one additional observation: If the branching variable x_h has $h \in S^a_0$, then the actual branching operation is skipped, as the feasible son obtained from the branching would be identical to the current node. The algorithm thus always chooses the minimum index in $N^a - S^a_0$ as the branching variable. No strict setting algorithm is used besides the setting obtained from Remark 2. Note that no cutting planes are used in any of the algorithms. While adding cuts would likely improve the results, they would muddy the comparisons between the different approaches for handling integer variables.

The application and the set of test problems are described briefly below. Table 1 gives characteristics of the test problems. Files of the test problems (in LP format) can be obtained from [17].

An *orthogonal array* with r runs, f factors, s levels, strength t and index λ is an $r \times f$ integer matrix whose entries are in $\{0, 1, \ldots, s-1\}$ and such that, in every $r \times t$ submatrix, each of the s^t possible distinct rows appears exactly λ times. The usual notation for such an array is $OA(r, f, s, t)$. (Note that as $r = \lambda s^t$, there is no need to record λ explicitly.)

Existence results and constructions for some values of the parameters can be found in [9]. In [1], the algorithms presented in this paper are used to enumerate all non isomorphic orthogonal arrays for some specified values of the parameters.

A simple set partitioning ILP formulation for these problems can be obtained by using a variable $0 \leq x_i \leq \lambda$ corresponding to the f-vector with entries corresponding to the representation of i in base s, for $i = 0, \ldots, s^f - 1$. Define a t-row F as any f-vector with entries in $\{0, 1, \ldots, s-1\}$ together with a set F_t of t indices in $\{1, \ldots, f\}$. The

Table 1
Problem characteristics; n: # variables; $\hat{z}$: optimal value, with $-$ denoting infeasible problems; LP: value of the LP relaxation of the initial formulation; Group order: order of the symmetry group

Problem	n	$\hat{z}$	LP	Group order
OA_2 (5, 3, 3, 2)	243	54	54	933,120
OA_2 (5, 3, 4, 2)	243	162	162	933,120
OA_3 (6, 2, 4, 3)	64	–	48	46,080
CA_3 (6, 2, 4, 3)	64	49	48	46,080
PA_3 (6, 2, 4, 3)	64	−44	−48	46,080
OA_5 (6, 2, 4, 3)	64	80	80	46,080
OA_2 (6, 3, 3, 2)	729	–	54	33,592,320
OA_5 (7, 2, 4, 5)	128	–	80	645,120
CA_5 (7, 2, 4, 5)	128	82	80	645,120
PA_5 (7, 2, 4, 5)	128	−76	−80	645,120
OA_7 (7, 2, 4, 7)	128	–	112	645,120
CA_7 (7, 2, 4, 7)	128	113	112	645,120
PA_7 (7, 2, 4, 7)	128	−108	−112	645,120
OA_6 (6, 2, 3, 4)	64	48	48	46,080
OA_7 (6, 2, 3, 4)	64	56	56	46,080
OA_6 (7, 2, 3, 3)	128	48	48	645,120
OA_7 (7, 2, 3, 3)	128	56	56	645,120
OA_6 (8, 2, 3, 3)	256	48	48	10,231,920

constraints of the ILP then require that for each fixed t-row F, the sum of the variables corresponding to f-vectors having all entries with indices in F_t identical to those in F should be equal to λ. As stated, the problem of determining if an orthogonal array exists or not is just a feasibility problem. Nevertheless, as packing and covering variants of the problem are also interesting, we work with the sum of the variables as objective function. We thus have a problem of the form:

$$\begin{aligned} &\min \quad 1^T x \\ &\text{s.t.} \quad Ax = \lambda, \\ &x \in \{0, \ldots, k\}^n. \end{aligned} \tag{5}$$

The value of k can of course be taken as λ, but sometimes it can be shown that a smaller value of k is enough. ILP (5) corresponding to the problem of constructing $OA(r, f, s, t)$ is denoted by $OA_\lambda(f, s, t, k)$. The covering version $CA_\lambda(f, s, t, k)$ (resp. packing version $PA_\lambda(f, s, t, k)$) is obtained by replacing the equality by $\geq$ (resp. $\leq$ and multiplying the objective function by -1). The symmetry group corresponds to the permutations of the columns and the permutations of the levels $\{0, \ldots, s-1\}$ for each factor of the orthogonal array and has order $(f!)(s!)^f$.

Table 1 lists characteristics of the test problems. The first six problems are fairly easy and can be solved quickly with commercial solvers. The next six problems are more difficult: Using *Cplex 9.0* [13] (using $\hat{z}-0.99$ as cutoff for packing or covering problems), OA_2 (6, 3, 3, 2) is not solved after 4 h, OA_5 (7, 2, 4, 5) is solved in 50 sec, CA_5 (7, 2, 4, 5) is solved in 2 h, OA_7 (7, 2, 4, 7) is solved in 45 min, CA_7 (7, 2, 4, 7) is solved in 2.5 h, and PA_7 (7, 2, 4, 7) is not solved after 4 h. The last five problems are easy, but we are interested in enumerating all their feasible non isomorphic solutions. No comparison with commercial solvers is possible.

ILP (5) can be transformed into a binary ILP by replacing each variable $0 \leq x_i \leq k$ by the sum of k binary variables $x_{i,1} + \cdots + x_{i,k}$. The number of variables is multiplied by k and the order of the symmetry group is multiplied by $(k!)$. This formulation is referred to as *Unary*. An alternative way to get a binary ILP is to replace $0 \leq x_i \leq k$ by $\lceil \log(k+1) \rceil$ variables with coefficients corresponding to powers of 2. For example, a variable $0 \leq x_i \leq 5$ is replaced by $4x_{i,1} + 2x_{i,2} + x_{i,3}$ everywhere in the ILP. The number of variables is multiplied by $\lceil \log(k+1) \rceil$, but the order of the symmetry group is unchanged. This formulation is referred to as *Log*. Note that, in the *Log* formulation, the constraints corresponding to the upper bounds $x_i \leq k$ are not included. It can be shown that, in the problems used

Table 2
CPU times (in seconds) and number of nodes for the three formulations

Problem	*Unary*		*Log*		*Int*	
	Time	Nodes	Time	Nodes	Time	Nodes
OA_2 (5, 3, 4, 2)	0.2	5	1.7	133	0.1	7
OA_2 (5, 3, 3, 2)	0.8	19	0.2	11	0.1	4
OA_3 (6, 2, 4, 3)	0.3	13	0.1	9	0.1	9
CA_3 (6, 2, 4, 3)	0.3	13	0.1	9	0.0	9
PA_3 (6, 2, 4, 3)	4.6	275	1.3	239	0.6	180
OA_5 (6, 2, 4, 3)	0.1	7	0.6	115	0.0	1
OA_2 (6, 3, 3, 2)	504.3	71	117.9	71	9.4	60
OA_5 (7, 2, 4, 5)	82.6	117	1.7	57	1.0	52
CA_5 (7, 2, 4, 5)	308.5	843	15.6	569	15.1	495
PA_5 (7, 2, 4, 5)	1574.5	5957	79.9	2307	75.6	2008
OA_7 (7, 2, 4, 7)	2136.7	1931	14.2	1417	9.0	764
CA_7 (7, 2, 4, 7)	2165.0	1931	22.5	1417	15.1	764
PA_7 (7, 2, 4, 7)	*	*	11,746.9	656,079	6874.4	357,946

A "*" is used for instances requiring more than a day of CPU time.

in the tests, they are always redundant, except for CA_5 (7, 2, 4, 5), OA_6(6, 2, 3, 4) and OA_7(6, 2, 3, 4) where their effect would be minimal. The *Unary* and *Log* formulations are solved using the branch-and-cut of [19] for comparison. The formulation obtained with ILP (5) is referred to as *Int* and is solved using the algorithms presented above.

Table 2 gives CPU times and number of nodes for the three formulations on the first 13 problems. Note that problem PA_7 (7, 2, 4, 7) with the *Unary* formulation can not be solved in a day of CPU time.

These results show that the *Int* formulation uniformly dominates the other two. Between the *Unary* and *Log* formulation, except on the second and sixth problems, the *Log* formulation is better than the *Unary* formulation. It is perhaps surprising to have the *Log* formulation as the worst on these problems, but two possible explanations come to mind: First, in the second problem we have that $k = 2 = \lceil \log(k+1) \rceil$ and thus the *Log* formulation has the same number of variables as the *Unary* formulation but the order of the symmetry group of the latter is larger. Second, the presence of non binary coefficients in the constraint matrix apparently makes the solution of the LP relaxations more fractional. Since these problems are feasible, the *Log* formulation enumerates more nodes before getting an LP relaxation that is integer.

On problems requiring more than 10 nodes, the *Int* formulation requires between 50% and 90% of the number of nodes for the *Log* formulation. Comparing the total CPU time, the *Int* formulation is up to 10 times faster (OA_2(6, 3, 3, 2)), but is about 2 times faster on most problems. The exceptions are CA_5(7, 2, 4, 5) and PA_5(7, 2, 4, 5) where performances are similar.

Note also that, for the *Int* formulation, the time spent handling the group operations is relatively small: For the first three problem, this amounts to less than 10% of the total CPU time and on the remaining problems, less that 2%. The only exceptions are PA_3 (6, 2, 4, 3) and OA_2 (6, 3, 3, 2) where this amounts to about 25% of the time. Note also that, in general, the LP relaxations obtained from the *Int* formulation are easier to solve or reoptimize than the ones obtained from the other two. For example, for problem CA_7 (7, 2, 4, 7), the average time per node spent for solving the LP relaxation is 0.0298 (*Unary*), 0.0112 (*Log*), and 0.0097 (*Int*). More strikingly, the average time spent per node for operations related to group operations is $1078.61 \cdot 10^{-4}$ (*Unary*), $9.67 \cdot 10^{-4}$ (*Log*), and $1.44 \cdot 10^{-4}$ (*Int*). Note that the factor 6 improvement between the *Log* and *Int* formulations likely comes from the additional fixing obtained using Lemma 6: Reducing entries in the date matrix leaves fewer possibilities to explore for the backtracking algorithms.

Table 3 lists the results for the enumeration of all feasible non isomorphic solutions of the last five problems of Table 1. The algorithms used are not particularly sophisticated and are obtained by a simple modification of the algorithms used above: The algorithms ignore integer solutions of the LP relaxation until all variables are fixed by branching (i.e. they enumerate all nodes in $\mathcal{T}$ in the notation of Section 3). When all variables are fixed, they write the current solution and do not update the value of the upper bound. These algorithms were used for solving several enumeration problems for orthogonal arrays [1]. The reader is referred to that paper for details. It can be observed that here also the *Int* formulation is uniformly better and that the time associated with group operations for the *Int*

Table 3
Time: CPU times in seconds; Iso: Time for operations related to group operations; Nodes: # nodes

Problem	*Log*			*Int*		
	Time	Iso	Nodes	Time	Iso	Nodes
OA_6 (6, 2, 3, 4)	6.8	3.3	3827	2.3	0.7	2331
OA_7 (6, 2, 3, 4)	15.1	5.4	10,433	5.0	1.0	5790
OA_6 (7, 2, 3, 3)	137.3	29.6	72,331	110.1	11.4	57,677
OA_7 (7, 2, 3, 3)	1964.77	186.2	1,104,599	1539.9	79.9	892,218
OA_6 (8, 2, 3, 3)	12,167.4	1267.7	3,008,337	10,385.8	637.5	2,651,934

Results for the enumeration of all non isomorphic feasible solutions.

formulation is roughly half the time for the *Log* formulation. Note that trying to solve any of these problems with the *Unary* formulation is close to hopeless due to the time taken by the group operations.

5. Coloring problems

This section is subdivided into four subsections, similar to those for Section 4 (the subsection treating basic group algorithms is not repeated). Section 5.1 deals with exclusion of some values for a variable. Sections 5.2 and 5.3 briefly cover the algorithms for computing orbits and testing if a set is a representative. Finally Section 5.4 presents a comparison between the branch-and-cut of [19], the branch-and-cut of Section 4 and the one developed in this section for solving edge coloring problems on graphs having a non trivial automorphism group.

In this section, we focus on coloring problems. These problems have the following structure: A set E of objects must be colored with k colors, respecting some coloring constraints. The ILP formulation used to model these problems is assumed to have k binary variables associated with each $e \in E$ with $x_{e,i} = 1$ if and only if e receives color i in the solution. We also assume that, in any feasible solution, each object must receive exactly one color.

Assuming that the objects in E can be permuted according to permutations in a group G, and assuming that all the k colors are interchangeable, the symmetry group of the ILP can be seen as the Cartesian product of G and the symmetry group Π^k, a group denoted by (G, Π^k). Its order is the order of G multiplied by $(k!)$. Since the group Π^k is simple enough to be handled efficiently without sophisticated data structure, the aim of this section is to develop algorithms working explicitly with G and implicitly with Π^k. The fact that Π^k is sometimes a worst-case example for group algorithms makes this idea attractive. Any permutation in the symmetry group of the ILP is the Cartesian product of a permutation $g \in G$ acting on the objects and a permutation $g_c \in \Pi^k$ acting on the colors. We denote it by (g, g_c), but this is a formal notation, as it is, in fact, just a permutation of the $|E| \cdot k$ variables of the ILP.

The data structure used for handling the group operations is similar to the one used in Section 4: For each object, we keep track of when a specific color for that object was excluded. This is done in an $|E| \times k$ date matrix. One major difference with Section 4 is that we do not treat one of the colors differently than the others. The fact that variables with value 0 were treated differently in Section 4 was justified by the usual structure of an optimal solution of combinatorial ILP. For coloring problems, its is usual that the colors are much more evenly distributed than the zeros and non zeros in the solution of an ILP. Moreover, the fact that we treat Π^k implicitly makes it necessary to remember things related to objects with color fixed to any of the colors, preventing us "skipping" objects of a particular color. We thus assume that the colors are numbered $\{1, \ldots, k\}$ and the possible "values" associated with an object is always a subset of these positive integers.

While the ILP works with binary variables, the group operations are performed as if, for each object $e \in E$, one integer variable with possible values in $\{1, \ldots, k\}$ is used. In particular, the branch-and-cut branches on an object $e \in E$, setting its color to all possible values. In the ILP, this means that it does not branch on a single variable $x_{e,i}$, but on all variables corresponding to object e simultaneously. We use the notations of Section 4, including F_p^a, f_p^a, D^a, and valsets (J, v) where $J \subseteq E$ and v is an $|E|$-vector with entries either -1 or in $\{1, \ldots, k\}$. For $g \in G$ and $g_c \in \Pi^k$ and a valset (J, v), $((g, g_c)(J), (g, g_c)(v))$ is the valset $(g(J), g(v))$ where the positive values of the entries in $g(v)$ are permuted according to g_c. Note that, in this section, $F_0^a = \emptyset$ for all a.

5.1. Setting variables

The results of Section 4.1 have an equivalent formulation for coloring problems. The proofs are so similar to those given earlier that they are skipped here.

Lemma 8. *In the statement of Lemma 4, replace the last two sentences by: Suppose that there exists* $(g, g_c) \in (G, \Pi^k)$ *with* $((g, g_c)(J), (g, g_c)(v)) \prec (F, f)$. *Then it is valid to set* $D^a[s, q] = |F_p^a|$.

Proof. Almost identical to the proof of Lemma 4. □

Note that Lemma 5 has no equivalent here, as F_0^a is always empty, implying that the condition $R^a[g(s)] < \bar{r}$ of that lemma can not be met.

Lemma 9. *In the statement of Lemma 6, replace* $g \in G$ *by* $(g, g_c) \in (G, \Pi^k)$ *and replace the last two lines by: If we have* $((g, g_c)((J \cup s) - \bar{s}), (g, g_c)(v)) = (F, f)$ *and* $D^a[g(\bar{s}), g_c(\bar{q})] \leq |J|$ *then it is valid to set* $D^a[s, q] = |F_p^a|$.

Proof. Almost identical to the proof of Lemma 6. It is even a little bit simpler, as we do not need to consider the case where (i_u, q_u) was obtained by fixing x_{i_u} to 0. □

5.2. Computing orbits in the stabilizer of a valset

The algorithm below consists of an initializing procedure cstabilizer_gen() that calls a recursive procedure cstab_gen(). This is the procedure stabilizer_gen of Section 4.3, slightly modified. The main difference is the presence of a k-vector *c_ord*, used to record the mapping of colors that have been made so far. If $c\_ord[c'] = c'' \neq -1$, then color c' is mapped to color c''.

Given $c', c'' \in \{1, \ldots, k\}$, we say that (c', c'') is *admissible* if either $c\_ord[c'] = c''$ or $c\_ord[c'] = -1$ and no entry in *c_ord* has value c''. In the former case, an object having color c' has already been sent on an object with color c''. In the latter case, no object with color c' has been processed yet and no object has been sent on an object with color c''.

For a given vector *c_ord*, a permutation $g_c \in \Pi^k$ is an *extension* of *c_ord* if it satisfies $g_c(c') = c\_ord[c']$ for all c' with $c\_ord[c'] \neq -1$. Observe that (c', c'') is admissible if and only if there exists an extension g_c of *c_ord* such that $g_c(c') = c''$.

The backtracking procedure given below outputs pairs of vectors $(perm, c\_ord)$ that can be used to construct generators of the stabilizer of the valset $(\beta[1..t], v)$ under (G, Π^k) for any integer vector v such that $v[\beta[j]]$ is the color associated with $\beta[j]$ for $j = 1, \ldots, t$ and -1 otherwise. To construct the generators associated with a pair $(perm, c\_ord)$, construct generators for the subgroup of (G, Π^k) permuting the objects according to *perm* and permuting colors according to any extension of *c_ord*. If z entries of *c_ord* are -1, exactly $\max\{1, z\}$ such generators are needed.

```
cstabilizer_gen(a, v, t)

/* Outputs pairs of vectors for building generators of
stab(β[1..t], v, (G, Π^k)) where G is the group represented by T with
base β */

   id_ord := k-vector with all entries -1;
   Set id_ord[v[β[i]]] = v[β[i]] for i = 1, ..., t;
   Output (p, id_ord) for all p ∈ U(β)_i, for i = t + 1, ..., n;
   ident := identity permutation;
   remain := β[1..t];
   c_ord := k-vector with all entries -1;
   cstab_gen(a, v, t, ident, c_ord, remain, 1);
```

The parameters of the call to cstab_gen() have the following interpretation: *ind* refers to the point $\beta[ind]$ being treated during the current call; *c_ord* is used to keep track of mapping of colors made so far. Note that $c\_ord[t] \neq -1$ if and only if t is the color associated with an object in $\beta[1..ind-1]$; *perm* is a permutation in G sending $\beta[1..ind-1]$ on a subset $B \subseteq \beta[1..n_f]$ with $c\_ord[v[\beta[i]]] = v[perm(\beta[i])]$ for $i = 1, \ldots, ind-1$ and *remain* is the set $perm^{-1}(\beta[1..n_f] - B)$.

```
cstab_gen(a, v, t, perm, c_ord, remain, ind)

  For each i ∈ remain do
    h := T[β[ind], i];
    If h ≠ ∅ then
      c := c_ord[f_p[β[ind]]];
      c' := f_p[perm(i)]];
      If (c, c') admissible then
         loc_c_ord := c_ord;
         loc_c_ord[c] := c';
         loc_remain := remain − i;
         loc_remain := h^{-1}(loc_remain);
         loc_perm := perm · h;
         If ind < t then
           cstab_gen(a, v, t, loc_perm, loc_c_ord, loc_remain, ind + 1);
         else
           output perm and loc_c_ord.
```

5.3. Deciding if a set is a representative or not

The following procedure tests if the valset $(\beta[1..n_f+1], v)$ is a representative of its orbit under (G, Π^k), where v is obtained from a copy of f_p and setting $v[\beta[n_f+1]] = q > 0$.

```
first_in_corbit(a, v)

/* Returns "true" if and only if (β[1..n_f + 1], v) is a
representative. */

   c_ord := k-vector with all entries -1;
   ident := identity permutation;
   remain := β[1..n_f + 1];
   is_lexmin := true;
   f_in_orb(a, v, ident, c_ord, remain, 1, is_lexmin);
   return(is_lexmin);
```

The Boolean parameter *is_lexmin* is passed by reference. As soon as *is_lexmin* = `false`, it is known that $(\beta[1..n_f+1], v)$ is not a representative and the procedure stops.

```
f_in_corb(a, v, perm, c_ord, remain, ind, is_lexmin)

  If is_lexmin = false then return;
  c := v[β[ind];
  For each i ∈ remain do
      h := T[β[ind], i];
      If h ≠ ∅ then
        c'' := v[perm(i)];
        If ∃ c' > c with (c', c'') admissible then (**)
          is_lexmin := false;
          return;
        If (c, c'') admissible then
          loc_c_ord := c_ord;
          loc_c_ord[c] := c'';
          loc_remain := remain − i;
          loc_remain := h^{-1}(loc_remain);
          loc_perm := perm · h;
          If ind < n_f then
            f_in_orb(a, v, loc_perm, loc_c_ord, loc_remain, ind + 1,
                                                          is_lexmin);
```

Lemma 10. *The algorithm first_in_corbit() is correct.*

Proof. Similarly to Lemma 7, if we define $B = perm(\beta[1..ind-1])$, then $B \subseteq \beta[1..n_f+1]$ and $perm(remain) = \beta[1..n_f+1] - B$.

Suppose that the algorithm returns `false`. Then, during a call at depth *ind* where (**) is satisfied, there exists an extension g_c of c_ord with $g_c[c'] = c''$. Applying the permutation $(h^{-1} \cdot perm^{-1}, g_c^{-1})$ to the set $B \cup perm(i)$ proves that the valset is indeed not a representative, as $perm(i)$ has color c'' and its image is $\beta[ind]$ with color c', a color larger than the color c of $\beta[ind]$.

If the valset is not a representative, there exists $J' \subseteq \beta[1..n_f+1]$ with corresponding color vector w', $\bar{g} \in G$ and $\bar{g}_c \in \Pi^k$ with $((\bar{g}, \bar{g}_c)(J'), (\bar{g}, \bar{g}_c)(w'))$ lexicographically smaller than the valset corresponding to $\beta[1 \cdots |J'|]$. If J' is chosen minimal with respect to inclusion with this property, there exists $j \in J'$ such that, for $J = J' - j$ and w the colors of the corresponding objects, we have that $((\bar{g}, \bar{g}_c)(J), (\bar{g}, \bar{g}_c)(w))$ is the valset corresponding to $\beta[1 \cdots |J|]$, that $\bar{g}(j) = \beta[|J|+1]$ and, for $c = v[\beta[|J|+1]$, $c'' := v[j]$ and $c' := \bar{g}_c[c'']$, we have $c' > c$. Then $\bar{g}$ will be the permutation $(h^{-1} \cdot perm^{-1})$ in a recursive call at depth $ind = |J|+1$, with the vector c_ord corresponding to the permutation of the colors induced by *perm* on the objects $\beta[1..ind-1]$. Since $\bar{g}_c^{-1}$ is an extension of c_ord, we have that (c', c'') is admissible and the algorithm will return `false`. □

Remark 4. It is possible to use Lemma 9 for modifying entries in D^a while running this algorithm: Just before the first block starting with "For each $i \in remain$ do", let $J' = perm(\beta[1..ind-1])$, $g := perm^{-1}$, and g_c^{-1} be any extension of c_ord.

- If $\beta[n_{f+1}] \in J'$ then take $s := \beta[n_{f+1}]$ and, for any $i \in remain$, let $\bar{s} := perm(i)$, $\bar{q} = f_p^a[\bar{s}]$, and $J := (J' \cup \bar{s}) - s$. Then $g(\bar{s}) = i$ and thus we can set $D^a[s,q] = |F_p^a|$ if $D^a[i, g_c[\bar{q}]] \leq |J| = ind - 1$ for some $i \in remain$.
- If $\beta[n_{f+1}] \notin J'$ then take $s \notin J', \bar{s} := s, q = \bar{q} > 0$ and $J := J'$. We can set $D^a[s,q] = |F_p^a|$ if $D^a[g(s), g_c[q]] \leq ind - 1$.

This is implemented in the code tested in Section 5.4. □

Remark 5. A remark similar to Remark 3 holds here too. □

5.4. Computational results

The machine and code specifications are similar to those described in Section 4.5. We report results for a few edge coloring problems on graphs having a non trivial automorphism group.

Table 4
Problem characteristics; vertices: # vertices; edges: # edges; Δ: maximum degree; Group order: Order of the automorphism group of the graph

Problem	Vertices	Edges	Δ	Group order
$of5\_14\_7$	14	35	5	48
$of7\_18\_9$	18	63	7	18,432
$ofsub9$	9	29	7	12
$jgt18$	18	33	4	2
$jgt30$	30	57	4	2
mered	70	140	4	38,698,352,640
*clique*9	9	36	8	362,880
*clique*9*p*	9	36	8	362,880
*clique*11*p*	11	55	10	39,916,800
*flosn*52	52	78	3	52
*flosn*60	60	90	3	60
*flosn*84	84	126	3	84
$O4\_35$	35	70	4	210

The minimum number of colors needed to color the edges of a graph G such that all edges adjacent to any vertex receive distinct colors is the maximum degree $\Delta(G)$ of a vertex in G. Vizing [24] proved that such a coloring always exists when the number of colors is larger than $\Delta(G)$, and graphs for which there is no coloring with $\Delta(G)$ colors are called *Class 2* graphs. The reader is referred to [25] for Graph Theory definitions and terminology.

We consider a few Class 2 graphs from the literature and use the branch-and-cut based on the algorithms described in this section to prove that they are indeed Class 2 graphs. The problems used in the test consist of:

- Three overfull graphs ($of5\_14\_7$, $of7\_18\_9$, and $ofsub9$);
- Two graphs from [6] ($jgt18$ and $jgt30$);
- The Meredith graph [21] (*mered*);
- The cliques on 9 nodes (*clique*9, *clique*9*p*) and 11 nodes (*clique*11*p*). The two versions for the clique on 9 nodes differ in the ordering of the edges: in *clique*9, a breadth-first ordering is used, while in *clique*9*p* the ordering is obtained by building the graph by adding repeatedly a node and edges to all existing nodes. The ordering for *clique*11*p* is similar to the one for *clique*9*p*;
- Three flower snarks [25] (*flosn*52, *flosn*60, and *flosn*84);
- The graph whose vertices correspond to the subsets of size 3 of a ground set of size 7, two vertices being adjacent if and only if the corresponding subsets are disjoint ($O4\_35$).

Characteristics of these graphs are listed in Table 4.

Using *Cplex 9.0* [13], half of these problems can be solved easily: $of5\_14\_7$ in 328 s, $of7\_18\_9$ instantly, $jgt18$ in 13 s, *clique*9 and *clique*9*p* instantly, *flosn*52 in 30 s, and *flosn*60 in 190 s. More difficult are *flosn*84 and $O4\_35$, requiring 2 h and 2.5 h respectively. The remaining four problems are not solved in 4 h. Note that preventing the solver from using cutting planes significantly changes the results only for four problems: $of7\_18\_9$, *clique*9, *clique*9*p*, and $O4\_35$ are then not solved in 4 h.

The ILP formulation for the edge coloring problem on graph $G = (V, E)$ used in the tests is the most basic one: For each edge $e \in E$ we have $\Delta(G)$ binary variables, with $x_{e,i} = 1$ if and only if color i is assigned to e. The constraints are from two families: First, for each vertex $v \in V$ and each color i, at most one edge incident to v receives color i. Second, for each edge e, exactly one color should be assigned to e. Files of the test problems (in LP format) and files describing the graphs can be obtained from [17].

We solve this ILP formulation first with the algorithm of [19] using the Cartesian product of the symmetry group of the graph and of the symmetry group of the colors as the symmetry group of the ILP. This algorithm is referred to as *Cart* in Table 5. We solve this same ILP formulation with the same symmetry group with the algorithm based on the results of Section 4. This algorithm is referred to as *Cart_Int* in Table 5. As the formulation is a binary formulation, the "integer" variables are, in fact, binary variables and using the algorithms of Section 4 might seem odd. This will be motivated when discussing the results below.

Table 5
CPU times (in seconds) and number of nodes for the three algorithms

Problem	*Cart*		*Cart_Int*		*Col*	
	Time	Nodes	Time	Nodes	Time	Nodes
*of*5_14_7	0.6	227	0.7	227	0.2	119
*of*7_18_9	27.9	2767	25.7	2733	17.0	1695
*ofsub*9	559.3	335,571	574.8	330,845	183.3	182,518
*jgt*18	3.6	3233	3.7	3233	1.2	1603
*jgt*30	855.4	497,401	902.7	497,401	274.4	248,729
mered	5.0	221	4.8	215	3.0	136
*clique*9	237.9	277	130.0	253	64.2	150
*clique*9*p*	0.7	75	0.6	75	0.2	43
*clique*11*p*	378.8	23,293	183.8	20,565	95.2	13,297
*flosn*52	8.6	1715	8.3	1683	6.4	929
*flosn*60	32.4	5413	31.7	5345	24.4	2973
*flosn*84	1984.4	208,737	1981.6	207,089	1488.0	116,439
*O*4_35	84.4	11,549	58.6	10,343	40.7	7209

We then solve the same ILP formulation, using the algorithms presented in this section, using the symmetry group of the graphs explicitly and the symmetry group of the colors implicitly. This algorithm is referred to as *Col* in Table 5.

The three algorithms are run with the minimum index branching rule, with the improvement mentioned in Section 4.5 for *Cart* and *Cart_Int*. No strict setting algorithm is used besides the setting obtained from Remark 2 for *Cart_Int* and from Remark 4 for *Col*. *Cart* uses a weaker version of the setting done by *Cart_Int*, corresponding to the first half of the setting described in Remark 2. As in Section 4.5, no cutting planes are used in any of the algorithms.

The comparison of the number of nodes for *Cart* and *Cart_Int* reflects the effect of the stronger setting. This effect is relatively small, less than 2% on most problems. The only exceptions are *clique*9, *clique*11*p* and *O*4_35 where the reduction is of the order of 10%. The CPU times are comparable, except for *clique*9, *clique*11*p* and *O*4_35 where *Cart_Int* is much faster. On the other problems, *Cart_Int* is a little bit slower on a per node basis in general. This should not be a surprise, since *Cart_Int* is designed to handle general integer problems, whereas *Cart* takes advantage of the fact that variables are binary. However, when the number of colors and group size increase, *Cart_Int* is usually faster as shown on *clique*9 *clique*11*p* and *O4_35*.

The comparison of the CPU time between *Cart_Int* and *Col* illustrates the benefits of handling the permutation group of the colors implicitly. While the improvement is not dramatic, most problems are solved with *Col* in 70% of the time used by *Cart_Int*. On two of the *of* problems and on the *jgt* problems this time is about 30% of the time used by *Cart_Int*, and on the *clique* problems it is about 50%. The difference is likely to be more significant for problems with a larger number of colors.

Comparing the number of nodes of *Cart_Int* with *Col* is tricky, since branching in *Cart_Int* amounts to fixing a binary variable to 0 or 1 while branching in *Col* amounts to fixing a color on an edge. If *Col* creates s sons and if the corresponding ILP are all feasible, then *Cart* or *Cart_Int* needs at most 2^s nodes to reach an equivalent situation. This is a very loose upper bound and a better one is the number of nodes in the binary tree corresponding to branching on s variables with at most two of the variables taking value 1 (a node with two variables set to one will be pruned by infeasibility). In addition, due to the way variables set to 0 are handled in *Cart_Int*, variables that are already set to 0 become fixed without having to actually branch, as indicated at the beginning of Section 4.5. Also, when some of the sons that *Col* creates are infeasible, it might happen that *Cart_Int* needs much less than 2^s nodes (and even, possibly, much less than s nodes) to reach an equivalent point.

As a side note, observe the stunning difference in CPU time needed for solving *clique*9 and *clique*9*p*. These two problems differ only in the ordering of the variables. Since a minimum index branching rule is used, the difference in time comes from a different choice of branching variables. In *clique*9, the chosen ordering implies that the symmetry between the colors tends to be destroyed as early as possible, whereas the opposite happens with the ordering for *clique*9*p*. This example shows that a careful study of the ordering of the variables when using the minimum index branching rule might transform an unsolvable problem into a manageable one.

Table 6
CPU times (in seconds) and number of nodes for *Cart_Int* and *Col* using a strict setting algorithm

Problem	*Cart_Int*		*Col*	
	Time	Nodes	Time	Nodes
*of*5_14_7	0.2	89	0.1	54
*of*7_18_9	20.2	2617	12.9	1524
*ofsub*9	130.3	105,653	70.9	58,797
*jgt*18	1.2	1049	0.8	621
*jgt*30	242.2	159,649	136.1	79,943
mered	3.6	147	2.7	93
*clique*9	124.7	249	60.1	139
*clique*9*p*	0.6	71	0.2	42
*clique*11*p*	184.2	20,561	93.8	13,273
*flosn*52	7.2	1677	5.1	927
*flosn*60	27.3	5339	19.2	2971
*flosn*84	1620.0	207,083	1176.6	116,437
*O*4_35	54.1	10,339	30.3	6605

To illustrate the benefits of using a strict setting algorithm, Table 6 gives the results obtained by *Cart_Int* and *Col* when they both use the following simple setting algorithm: When the color of edge e is fixed or set to c, that color is immediately excluded for all edges $e' \neq e$ sharing an endpoint with e. In the LP, all variables $x_{e,i}$ for $i \neq c$ are set to 0 as well as all variables $x_{e',c}$ for all edges e' sharing an endpoint with e. Note that these setting operations do not strengthen the LP bound directly, as the constraints of the ILP already imply them as soon as $x_{e,c} = 1$ is in force. However, settings obtained from Lemma 9 may become more efficient as some entries in D^a are reduced.

Comparing results for *Cart_Int* and *Col* in Tables 5 and 6, the number of nodes is divided by a factor 3 on the first five problems, reduced by about 30% on *mered* and virtually inexistant on the remaining problems.

The CPU time for *Cart_Int* is divided by a factor of 3 on the first five problems, reduced by about about 30% on *mered* and by about 10%–20% on the *flosn* problems and on *O4_35* and virtually nonexistent on the *clique* problems. The CPU time for *Col* is divided by a factor of 2 on the first five problems, reduced by about about 10% on *mered* and by about 20% on the *flosn* problems and on *O4_35* and virtually nonexistent on the *clique* problems.

Here also, *Col* is faster than *Cart_Int* on all problems. Note that the fraction of the CPU time spent for handling the group operations is relatively modest, at around 7% for both algorithms (with the exception of *clique*9 where it is 99%, *clique*11*p* where it is 30%, and *O*4_35 where it is 20%).

These results show that strict setting algorithms might play a crucial part in solving symmetric ILP. Despite the fact that the setting used above is useless for improving the LP bound, the additional setting that is obtained using Lemma 9 produces a significant reduction in the number of nodes in some problems, and a significant reduction in CPU time on most problems. This also suggests that finding practical implementations of Lemma 9, extending the settings that can be done in a reasonable amount of time, might improve the solution time significantly.

References

[1] D.A. Bulutoglu, F. Margot, Classification of orthogonal arrays by integer programming, 2005. Working paper.
[2] G. Butler, Computing in permutation and matrix groups II: Backtrack algorithm, Mathematics of Computation 39 (1982) 671–680.
[3] G. Butler, Fundamental Algorithms for Permutation Groups, in: Lecture Notes in Computer Science, vol. 559, Springer, 1991.
[4] G. Butler, J.J. Cannon, Computing in Permutation and matrix groups I: Normal closure, commutator subgroups, series, Mathematics of Computation 39 (1982) 663–670.
[5] G. Butler, W.H. Lam, A general backtrack algorithm for the isomorphism problem of combinatorial objects, Journal of Symbolic Computation 1 (1985) 363–381.
[6] A.G. Chetwynd, R.J. Wilson, The rise and fall of the critical graph conjecture, Journal of Graph Theory 7 (1983) 153–157.
[7] http://www.coin-or.org.
[8] M. Elf, C. Gutwenger, M. Jünger, G. Rinaldi, Branch-and-cut algorithms for combinatorial optimization and their implementation in ABACUS, pp. 155–222. In [10].
[9] A.S. Hedayat, N.J.A. Sloane, J. Stufken, Orthogonal Arrays: Theory and Applications, Springer, 1999.
[10] M. Jünger, D. Naddef (Eds.), Computational Combinatorial Optimization, in: Lecture Notes in Computer Science, vol. 2241, Springer, 2001.

[11] D.F. Holt, B. Eick, E.A. O'Brien, Handbook of Computational Group Theory, Chapman & Hall/CRC, 2004.
[12] C.M. Hoffman, Group-Theoretic Algorithms and Graph Isomorphism, in: Lecture Notes in Computer Science, vol. 136, Springer, 1982.
[13] ILOG CPLEX 9.0 User's Manual, 2003.
[14] D.L. Kreher, D.R. Stinson, Combinatorial Algorithms, Generation, Enumeration, and Search, CRC Press, 1999.
[15] J.S. Leon, On an algorithm for finding a base and a strong generating set for a group given by generating permutations, Mathematics of Computation 35 (1980) 941–974.
[16] J.S. Leon, Computing automorphism groups of combinatorial objects, in: M.D. Atkinson (Ed.), Computational Group Theory, Academic Press, 1984, pp. 321–335.
[17] http://wpweb2.tepper.cmu.edu/fmargot/index.html.
[18] F. Margot, Pruning by isomorphism in branch-and-cut, Mathematical Programming 94 (2002) 71–90.
[19] F. Margot, Exploiting orbits in symmetric ILP, Mathematical Programming. Series B 98 (2003) 3–21.
[20] D. McKay, Isomorph-free exhaustive generation, Journal of Algorithms 26 (1998) 306–324.
[21] G.H.J. Meredith, Regular n-valent n-connected nonHamiltonian non-n-edge-colorable graphs, Journal of Combinatorial Theory. Series B 14 (1973) 55–60.
[22] J.H. Owen, S. Mehrotra, On the value of binary expansions for general mixed-integer linear programs, Operations Research 50 (2002) 810–819.
[23] M.W. Padberg, G. Rinaldi, A branch-and-cut algorithm for the resolution of large scale symmetric travelling salesman problems, SIAM Review 33 (1991) 60–100.
[24] V.G. Vizing, On an estimate of the chromatic class of a p-graph, Diskretnyĭ Analiz 3 (1964) 25–30.
[25] D.B. West, Introduction to Graph Theory, 2nd edition, Prentice Hall, 2001.
[26] L.A. Wolsey, Integer Programming, Wiley, 1998.